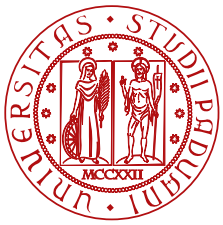

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

A Bytecode Virtual Machine

Relatore
Prof. Silvestri Francesco

Laureando
Battisti Filippo
Matricola N. 2066659

ANNO ACCADEMICO 2024-2025

22/09/2025

to my grandfather, whose creativity is and always will be
my source of inspiration.

Abstract

In the modern software development landscape, the demand for rapid development often outweighs concerns about low-level performance, leading to widespread adoption of platform-independent languages. Cross-platform compatibility is achieved through the use of virtual machines, which interpret intermediate bytecode rather than compiling directly to machine code. This thesis presents the design and implementation of a lightweight, cross-platform virtual machine tailored for executing procedural programs in terminal environments. Drawing conceptual inspiration from established virtual machine architectures, the project explores low-level system design, bytecode interpretation, and memory management. A key contribution is the development of a custom instruction set architecture, designed to reflect conventional processor principles while remaining compact and extensible. This ISA supports fundamental operations such as arithmetic, control flow, and memory access, enabling efficient execution of moderately complex programs. While the current implementation has limitations, such as restricted data structure support and constrained function scope, it provides a solid foundation for further development. The work highlights critical trade-offs in virtual machine design and contributes toward the creation of a platform-independent procedural programming language.

Contents

Chapter 1

Introduction

In today's software development landscape, time is a crucial factor. Companies often prioritize development speed over performance, which is why programming languages like Python and JavaScript are so widely used in the industry. These languages adhere to the ”Write once, run anywhere” philosophy, a slogan coined in 1995 by Sun Microsystems to showcase the versatility of the Java programming language. However, in the early days, this promise often fell short in practice, giving rise to the ironic twist: “Write once, debug everywhere.” Inconsistent behavior across platforms made cross-platform development more challenging than anticipated. Fortunately, after years of advancements, modern cross-platform systems have become significantly more reliable, bringing the original ideal closer to reality.

1.1 Process Virtual Machine

One of the key features that makes certain programming languages platform independent is the use of a *Virtual Machine* (VM). Instead of compiling code directly into machine-specific instructions, these languages compile down to an intermediate form called *bytecode*. This bytecode is then executed by a process virtual machine, a program that simulates a processor in software.

By using a virtual processor, the same bytecode can run on any system that supports the corresponding virtual machine, regardless of the underlying hardware or operating system. This is what allows languages like Java and C# to be platform independent.

To improve performance, many virtual machines include a *Just-In-Time* (JIT) compiler, which translates bytecode into native machine code at runtime. This means frequently executed code can be optimized on the fly, blending the portability of virtual machines with near native execution speed.

Modern virtual machines like the Java Virtual Machine (JVM) or the .NET Common Language Runtime (CLR) handle not only execution but also memory management, security, and performance optimizations, making them powerful and flexible platforms for running software across diverse environments.

1.2 Java

One of the most influential programming languages of all time is Java, created in the 1990s by James Gosling. Over the years, it became an industry standard, so much so that many of the programs we use daily are built with Java. As mentioned earlier, Java is a platform-independent language. Unlike other languages, its source code is not compiled directly into machine instructions. Instead, it is compiled into bytecode, which runs on the JVM. However, Java is not the only language that runs on the JVM. Over time, many other languages have been developed to run on it, including Kotlin, Scala, and Groovy, just to name a few. A significant portion of the design choices for my project are inspired by the Java Virtual Machine. As a result, throughout the exploration of the virtual machine's structure, I will frequently compare my design decisions to those of the JVM. The primary sources I have used to learn about the JVM structure include The Java Virtual Machine Specification (Java SE 8 Edition) [4] and the open source JDK GitHub repository. Another valuable resource has been .NET IL Assembler [3], which explains the structure of the Common Language Runtime (CLR), the virtual machine developed by Microsoft for the .NET framework.

1.3 Raiu

The motivation behind designing a custom virtual machine arose from the goal of creating a new programming language: *Raiu*. Over the years, through extensive experience with object-oriented programming (OOP), I've become increasingly aware of its limitations, particularly its tendency to model software in a way that reflects how humans perceive the world, rather than how computers operate.

Humans naturally think in terms of "things" (objects) that have properties (attributes) and behaviors (methods), which aligns well with the object-oriented paradigm. However, this abstraction can introduce unnecessary complexity and inefficiency when building systems that require low-level control or performance.

In contrast, I have a stronger preference for *procedural programming* (PP), which more closely reflects the logic of computer execution: linear control flow, explicit state changes, and a clearer

mapping to how machines operate.
The Raiu language, named as an anagram of the Indonesian archipelago “Riau”, draws inspiration from C, but will introduce several modern features such as *Resource Acquisition Is Initialization* (RAII), safe buffer access, generics, and more; while being, through a virtual machine, independent on the underlying hardware and operating system.

1.4 Objectives

This thesis makes two primary contributions:

- **Raiu Virtual Machine (RVM).**
  A lightweight, stack-based VM for terminal applications that minimizes abstraction of memory layout and resource management, aligning closely with hardware to maximize performance.

- **Raiu Instruction Set (RIS).**
  A compact, extensible ISA that mirrors real processor conventions, load-store, arithmetic, control flow, while remaining simple enough for fast execution.

The thesis is organized in two main parts.
First (Chapter 2), the RVM’s internal architecture is explored: its data and memory models, call stack layout, constant pools, and interpreter structure.
Then (Chapter 3), the RIS is presented in full detail: its encoding rules, instruction families, and example mappings from high-level constructs to bytecode, enabling readers to understand and design their own language to run on the RVM.

Chapter 2

Raiu Virtual Machine

In this chapter, the structure of the Raiu Virtual Machine (RVM) is examined from the ground up. The discussion begins with its most atomic elements; primitive data types, memory layouts, and the smallest execution units, and then progressively assemble these into higher-level constructs such as call frames, constant pools, and module tables. Finally, The chapter concludes showing how these components fit together in the interpreter's execution loop and loading pipeline. By starting small and gradually expanding our view, the reader will gain a clear, layered picture of the RVM's internal architecture.

2.1 Types

The RVM supports a limited set of primitive types, meaning that these types can be stored in variables, passed as arguments, returned by functions, and operated upon. However, the virtual machine does not track the actual type of the data. Instead, it is instructed to interpret the values according to their intended type. Ensuring the correct usage of types is expected to be handled prior to runtime, typically by a compiler.

2.1.1 Super Types

To perform general operations on data, I have defined four supertypes that represent memory blocks of specific sizes. The RVM is natively instructed to operate on these types, which are as follows:

- `byte` : for 8-bit or 1 byte values
- `hword` : for 16-bit or 2 byte values
- `word` : for 32-bit or 4 byte values

- `dword` : for 64-bit or 8 byte values

For these types, the VM performs load/store operations at specific memory addresses and manipulates bits with bitwise operations.

2.1.2 Integral Types

The native integral types are divided into four size groups, with their signed or unsigned variant:

- `i8` : a byte sized signed integer whose values range from -2^7 to $2^7 - 1$
- `u8` : a byte sized unsigned integer whose values range from 0 to $2^8 - 1$
- `i16` : a half word sized signed integer whose values range from -2^{15} to $2^{15} - 1$
- `u16` : a half word sized unsigned integer whose values range from 0 to $2^{16} - 1$
- `i32` : a word sized signed integer whose values range from -2^{31} to $2^{31} - 1$
- `u32` : a word sized unsigned integer whose values range from 0 to $2^{32} - 1$
- `i64` : a double word sized signed integer whose values range from -2^{63} to $2^{63} - 1$
- `u64` : a double word sized unsigned integer whose values range from 0 to $2^{64} - 1$

Unsigned integers are represented using the classical binary format. For a given number of bits n, and a sequence of bits $x = \{x_1, x_2, \ldots, x_n\}$, where each $x_i \in \{0, 1\}$, the value of the unsigned integer is calculated as:

$$y_u(x) = \sum_{i=0}^{n-1} x_i \cdot 2^i$$

For signed integers, the binary two's complement representation is used [6]. Given the same bitstring x of n bits, the corresponding signed integer value is calculated as:

$$y_s(x) = -x_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i$$

Here, x_{n-1} is the most significant bit and serves as the sign bit. This representation allows for an integers range from -2^{n-1} up to $2^{n-1} - 1$, and enables efficient binary arithmetic, including addition, subtraction, and negation without requiring special handling for the sign bit.

2.1.3 Floating Point Types

The supported floating types are the 32-bit and the 64-bit formats described in the IEEE 754 Standard for Binary Floating-Point Arithmetic [1], those two types are:

- `f32` : the 32-bit single precision floating point from $-(2-2^{-23})\times2^{127}$ to $(2-2^{-23})\times2^{127}$
- `f64` : the 64-bit double precision floating point from $-(2-2^{-51})\times2^{1023}$ to $(2-2^{-51})\times2^{1023}$

Before explaining the IEEE standard format, we have to define some useful constants :

- N : is the bit count of the type and can be 32 or 64
- M : is the amount of bits that describes the number's mantissa, it is 23 for `f32` values and 51 for `f64` values
- E : is the amount of bits that describes the number's exponent $E = N - M$
- e_b : is the bias exponent, it is 127 for `f32` values and 1023 for `f64` values

Floating point numbers are represented as

$$v = (-1)^s \times (1 + \frac{m}{2^{M-1}}) \times 2^{e-e_b} \tag{2.1}$$

where

- s : is the most significant bit (bit at index N-1)
- e : is the unsigned integer derived from bits $[M, N-1)$
- m : is the unsigned integer derived from bit $[0, M)$

This model allows for the representation of a wide range of numbers, from $-(2-2^{-M})\times2^{e_b}$ to $(2-2^{-M})\times2^{e_b}$. There are some unique cases where the formula (2.1) is not a valid representation, those cases are:

- $e = 2^E - 1 \wedge m \neq 0$: the value represented is NaN (Not a Number)
- $e = 2^E - 1 \wedge m = 0$: the value represented is $(-1)^s \times \infty$
- $e = 0 \wedge m = 0$: the value represented is $(-1)^s \times 0$

2.1.4 Reference Types

In the RVM, references are implemented as raw 64-bit pointers. This approach, similar to that used by the CLR, allows for direct memory access without an intermediate layer of indirection or metadata. Each reference is simply a native memory address, which simplifies the implementation of memory management operations. The directness of this model enables faster access times and reduces the overhead typically associated with more abstract reference representations.
This design inherently restricts the VM to 64-bit architectures, as references are assumed to be 64 bits wide. However, this limitation is acceptable in practice, given the prevalence of 64-bit systems in modern computing environments. Additionally, because references are native pointers, variables can, in principle, be referenced even when allocated on the stack, a flexibility not typically available in more abstract virtual machines.

2.1.5 Boolean Type

I have chosen not to define a separate boolean type for the RVM, despite supporting conditional operations, as detailed in Section 3.10. Instead, a boolean expression is represented using a 32-bit integer, where:

- A value of zero represents False.
- Any non-zero value represents True.

Although booleans are not explicitly defined as a native type in this implementation, a hypothetical compiler could easily implement them using an 8-bit integer, as is common in many modern programming languages.

2.1.6 Buffers

Buffers are contiguous blocks of memory used to store elements of the same size. The RVM provides dedicated instructions for handling such structures.
A buffer may reside either on the stack or in the heap. It is referenced using a pointer to its first element. Preceding each buffer in memory, the RVM stores a 64-bit unsigned integer representing the buffer's total capacity in bytes. This metadata allows the runtime to perform bounds checking and ensure that accesses remain within the buffer's allocated space.

```
buffer[] = { u64 size, elem0, elem1, ..., elemN }
```

2.1.7 String Type

Strings are sequences of characters, and in this implementation, they are represented using ASCII-encoded characters, each stored in a single byte. As with booleans, there is no dedicated string type; however, due to their underlying integer representation, string manipulation operations remain feasible. In line with the approach taken by languages such as Java, the RVM treats strings as character buffers. This design choice improves safety compared to null-terminated strings used in languages like C or C++, reducing risks such as buffer overflows and accidental memory corruption.

2.1.8 Type Grouping

A common feature in modern programming languages is the ability to group different types into more complex data structures. The elements contained within such structures are typically referred to as members or fields.

In many high-level virtual machines, such as the JVM or the CLR, field access is typically managed using metatables or metadata structures that store layout information, such as field offsets and sizes. While this approach allows for greater flexibility, it does not scale well with deeply nested structures, as the time complexity for accessing a field grows linearly with the nesting depth.

To avoid this overhead, the RVM design encodes field offsets directly into the bytecode. This is achieved using a set of dedicated instructions for offset-based loads and stores (see Section 3.13.3). This strategy eliminates the need for runtime metadata lookups and enables constant-time field access. However, it requires all field offsets to be resolved at compile time.

One current limitation of type grouping in this architecture is that the maximum size of a structure is limited to 256 words (1 kB), due to the encoding constraints of the instruction set.

2.2 The Call Stack

Similar to the JVM and the CLR, the RVM is a stack-based machine. Stacks are particularly well-suited for passing arguments that are needed for immediate manipulation due to their First-In Last-Out (FILO) property.

The call stack, analogous to the C runtime stack, is implemented as a contiguous block of memory, specifically 2^{20} words (4MB), that grows upwards and consists of stack frames placed one on top of another.

Each time a function is invoked, a new stack frame is pushed onto the stack. A stack frame

contains several elements in the following order:

1. A reference to the previous stack frame.
2. A reference to the next instruction to execute when returning from the function.
3. A reference to the header of the caller function (See Section 2.4).
4. A fixed-size word array containing the local variables.
5. A fixed-size word stack used to pass instruction arguments, referred to as the *Operation Stack*.

Whenever a new scope is pushed onto the stack, its validity is checked. If the scope exceeds the memory allocated for the stack, a "StackOverflowError" message is printed to the console, and the program terminates. This limitation is common across many modern programming environments. However, in the future, it may be possible to implement a setting within the virtual machine that allows users to define a custom stack size.

2.2.1 Local Variables

Each function in the RVM stores its local variables in a fixed-size array of word-sized cells. The array has a maximum capacity of 256 words, equivalent to 1 kB of total local storage.
This design imposes a strict upper limit on stack-based storage. Consequently, when working with larger data structures such as arrays or composite types, the programmer must take this constraint into account. Data structures exceeding 1 kB in size cannot reside on the stack and must be allocated on the heap instead.
A notable improvement in this architecture over more abstract virtual machines, such as the JVM, is its efficient use of local storage. Thanks to the instruction set (see Section 3.3), multiple local variables can be packed into a single word if their sizes allow it. For example, unlike the JVM, which requires local variables to be aligned to 4 bytes (or even 8 bytes in certain scenarios), this design permits tighter packing of smaller types like `byte` and `short`, which only require 1 or 2 bytes respectively.

2.2.2 Operation Stack

Functions in the RVM are equipped with a local stack, referred to as the *Operation Stack*. This component serves a role analogous to that of a register set in assembly languages, as it is directly involved in the execution of operations.

The decision to adopt a stack-based instruction set, rather than a register-based one, stems from the objective of designing an 8-bit instruction format. In traditional assembly languages, each instruction must explicitly specify the registers involved in the operation, which increases the instruction size. In contrast, with a stack-based model, this information is implicitly managed through the Operation Stack, thereby reducing instruction complexity and size.
However, this approach introduces a trade-off. In a stack-based system, operands are not preserved after use, since each operation consumes its operands from the stack. To solve this problem, there exists a family of instructions to duplicate stack data.

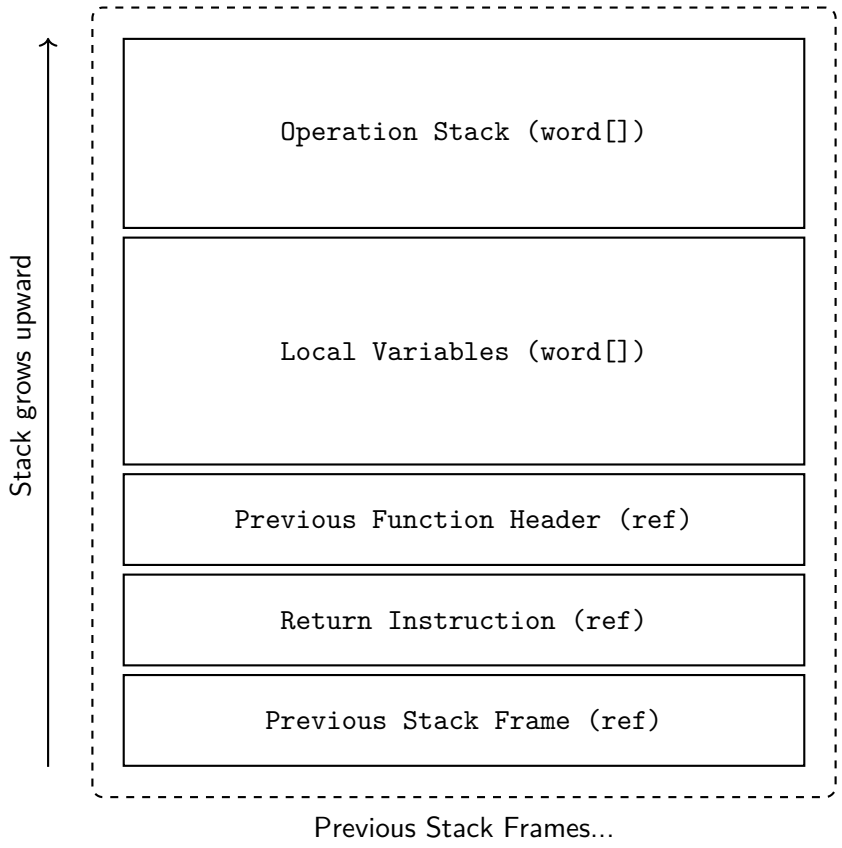

Figure 2.1: Stack Frame Layout

2.3 Constant Pools

Constant pools contain several kind of constant values that are required at runtime. A constant pool serves the scope of storing constant data, too big to be stored in the bytecode, that is needed during runtime.
There are five constant pools during the RVM runtime.

- Word constant pool : contains 32-bit integers and 32-bit floating point values.
- Double-Word constant pool : contains 64-bit integers and 64-bit floating point values.
- String constant pool : contains a set of references to ASCII character buffers.
- Global variables pool : contains a set of references to global variables.
- Function constant pool : contains a set of references to the funcion headers.

The word and double-word constant pools are accessed by retrieving a copy of the value contained in the accessed index. The string pool instead of copying the value returns a pointer to the first character of the string; this approach was chosen because string copying is a performance intensive operation; the downside of this method is that a possible corruption of the original string, that is supposed to be immutable, is possible; therefre, the responsibility of the correct string management is left to the compiler.

2.4 Functions

Functions are blocks of code dedicated to performing specific tasks. As described in Section 2.2, every time a function is invoked, a new stack frame is pushed onto the call stack. The information required invoke and terminate a function is contained in its function header, a block of memory located at the function's entry point. The function header includes the following fields:

- Module Table: A reference to the module table (see Section 2.5).
- Signature: A reference to the function's signature string (see Section 2.6).
- Argument Word Count (AWC): A 16-bit unsigned integer specifying how many words are passed as arguments from the caller's Operation Stack.
- Local Word Count (LWC): A 16-bit unsigned integer indicating the number of words allocated for the local variable array.
- Stack Word Count (SWC): A 16-bit unsigned integer defining the maximum size of the Operation Stack for this function.
- Return Word Count (RWC): A 16-bit unsigned integer indicating the number of words to return after function execution.

When a function is called, a number of words equal to the AWC are popped from the caller's Operation Stack and copied into the callee's local variables array starting at index 0. After the function completes execution, a number of words specified by the RWC are popped from the callee's operation stack and pushed onto the caller's operation stack.
It is important to note that the AWC must always be less than or equal to the LWC. This ensures that arguments can be stored safely in the local variables array.
The function body begins immediately after the header, and subsequent functions are laid out contiguously in memory, each aligned to 8 bytes.

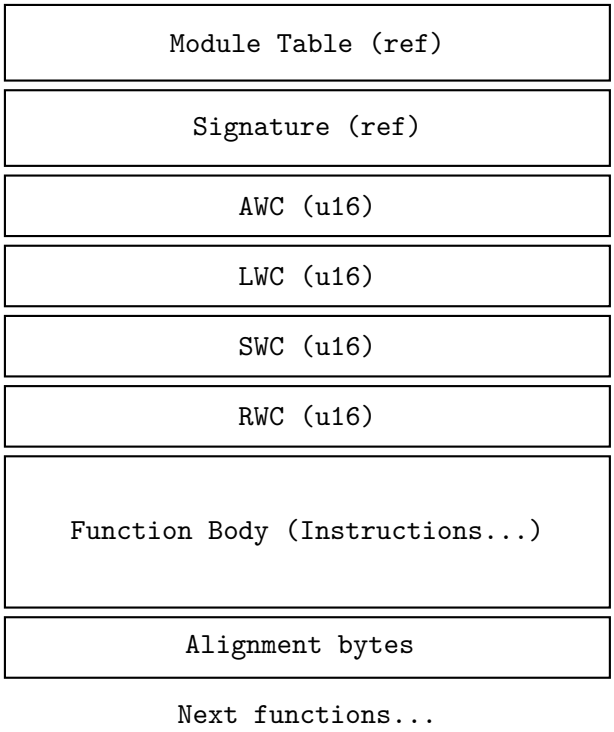

Figure 2.2: Function Layout

2.5 Modules

The RVM does not rely on a single, monolithic executable file. Instead, similar to the design of the JVM, it operates using multiple files known as *modules*.
Each module encapsulates both executable instructions and the metadata required by the virtual machine to interpret and run the code. This modular architecture helps overcome internal limitations, such as the maximum number of constants and functions per module, which are capped at 2^{16}.
At runtime, each module is converted into a *module table*, which allows the interpreter to ac-

cess constant pools relative to the currently executing function. Module tables do not own the memory that stores application buffers; they contain only references to the data, accessed either *directly* or *indirectly*. For example, word and double-word pools are accessed through direct pointers, since these types have a fixed size. In contrast, strings, functions, and global variables, due to their dynamic sizes, must be accessed through indirect buffers that store references to each element in their respective pools.

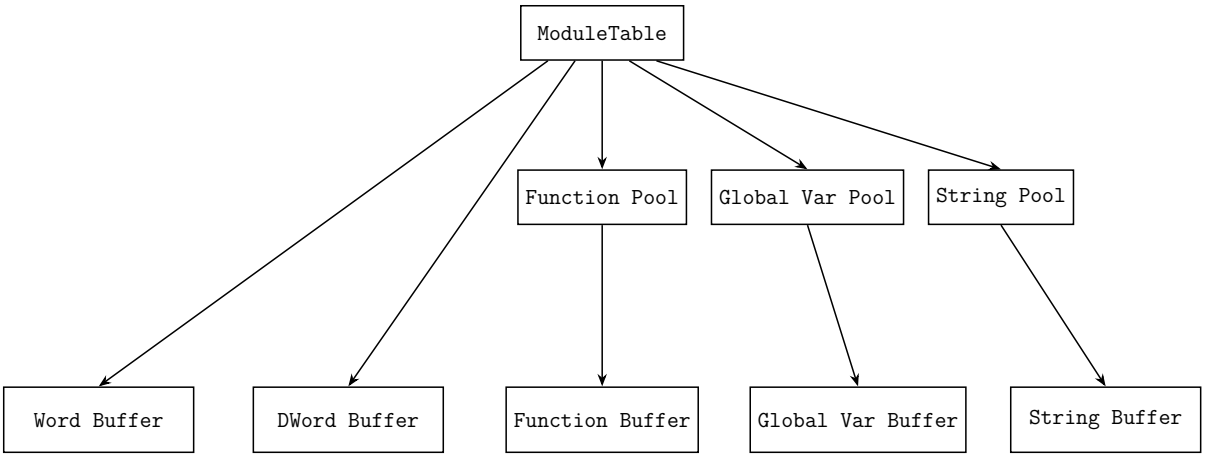

Figure 2.3: Module Table

2.6 Signatures

To enable seamless communication between modules, all functions and global variables are uniquely identified by a *signature*. A signature is a string that encodes the full path from the package root to the module file, followed by the symbol name.
This naming scheme ensures global uniqueness and prevents conflicts between modules.

```
string: MyPackage/Folder1/…/File.FunctionName
string: MyPackage/Folder1/…/File.VariableName
```

2.7 Module Files

The first ten bytes of a module file store the sizes of various constant pools as 16-bit unsigned integers in the following order:

1. `WordPoolSize`: Number of entries in the word constant pool.
2. `DWordPoolSize`: Number of entries in the double-word constant pool.

3. `StringPoolSize`: Number of strings in the string constant pool.
4. `GlobalPoolSize`: Number of global variable declarations.
5. `FunctionPoolSize`: Number of function declarations.

Next, the executable stores the constant pools themselves:

- An array of words of size `WordPoolSize`.
- An array of double-words of size `DWordPoolSize`.
- A string pool consisting of a sequence of null-terminated ASCII strings, with the total number of strings given by `StringPoolSize`.

For global variables and function declarations, the data is divided into two parts: *declarations* and *definitions*. Declarations are stored as character arrays using the same null-terminated string format. Two arrays follow:

- Global variable signatures.
- Function signatures.

Signatures can be either *internal* or *external*. Internal signatures consist only of the variable or function name, while external signatures represent references to symbols declared in other modules. In the bytecode, internal signatures appear first. If external variables or functions are referenced, they are separated from internal declarations by a special marker: the string ”&” (an ampersand character followed by a null terminator).

```
string[] : { "IntVar0", "IntVar1", …, "&", "PackageX/….ExtVar0", …}
string[] : { "IntFun0", "IntFun1", …, "&", "PackageY/….ExtFun0", …}
```

After the declarations, the binary stores the definitions. For global variables, the module includes an array of 32-bit unsigned integers describing the size of each variable. The length of this array matches the number of internal global variable declarations.
For function definitions, each function starts with a 32-bit unsigned integer representing the size of the function body in bytes. This is followed by eight bytes containing:

1. AWC as a 16-bit unsigned integer.
2. LWC as a 16-bit unsigned integer.
3. SWC as a 16-bit unsigned integer.
4. RWC as a 16-bit unsigned integer.

The function body is stored as a sequence of bytecode instructions, whose size is determined by the previously specified body size. The number of function definitions is determined by the number of internal function declarations.

```
ModuleFile
{
    u16 WordPoolSize
    u16 DWordPoolSize
    u16 StringPoolSize
    u16 GlobalPoolSize
    u16 FunctionPoolSize

    Word  ConstantWords[WordPoolSize]
    DWord ConstantDWords[DWordPoolSize]
    char  ConstantStrings[...]

    # Global variable and function declarations
    char  GlobalSignatures[...]
    char  FunctionSignatures[...]

    # Global variable and function definitions
    u32 GlobalSizes[GlobalPoolSize]

    u32 BodySize_0
    u16 AWC_0
    u16 LWC_0
    u16 SWC_0
    u16 RWC_0
    u8 Body_0[BodySize_0]
```

```
    ... # Additional functions
}
```

2.8 The Interpreter

The component of the virtual machine responsible for executing programs is the *interpreter*. Its sole task is to read bytecode instructions and execute them sequentially, one at a time.
To operate effectively, the interpreter uses a set of internal registers referred to as `Control Flow Registers` (CFRs). These registers manage the state of the program during execution:

- `OP` (Opcode): Holds the opcode of the currently executing instruction.
- `PC` (Program Counter): Points to the next instruction to fetch from memory.
- `SP` (Stack Pointer): Points to the top of the Operation Stack.
- `FP` (Frame Pointer): Points to the base of the current stack frame.
- `FH` (Function Header): Reference to the current function header.
- `WPool` (Word Pool): Pointer to the current word pool.
- `DPool` (Double-Word Pool): Pointer to the current double-word pool.
- `SPool` (String Pool): Pointer to the current string pool.
- `GPool` (Global Variable Pool): Pointer to the current global variable pool.
- `FPool` (Function Pool): Pointer to the current function pool.

Instruction dispatch in the interpreter is very simple: the byte at the address held in the `PC` register is fetched as the current opcode, and a jump table is used to branch directly to the corresponding handler label, where the opcode's semantics are implemented.
Studying the OpenJDK HotSpot source was extremely valuable: it demonstrates how a *Direct Threaded Interpreter* can be built using GCC's *computed goto* feature to jump straight to label pointers instead of relying on a *switch-case dispatch*. This technique reduces branch misprediction and overhead, yielding noticeably better performance than a traditional switch-based interpreter.
The HotSpot virtual machine further improves interpreter throughput using *Bytecode Template*

Interpretation. In this approach, HotSpot generates small fragments of native code, one per opcode, that are optimized for the target architecture. At runtime, the interpreter simply jumps into a contiguous “template” buffer of these precompiled handlers, combining the compactness of bytecode with the speed of directly executed machine code.

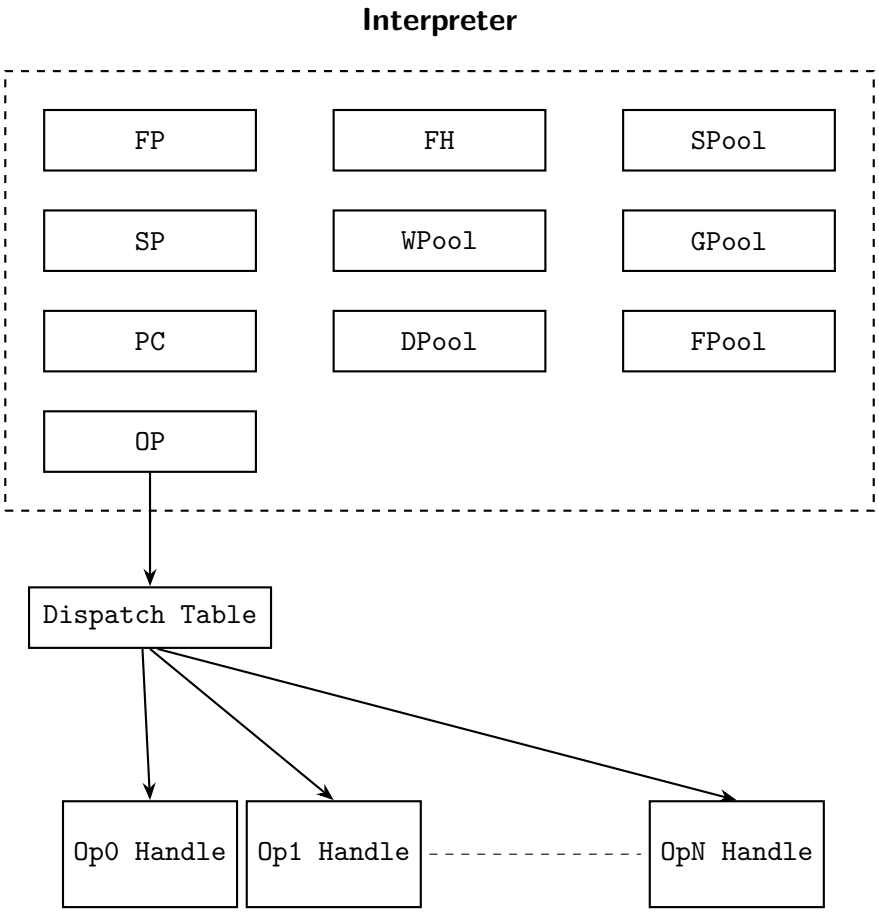

Figure 2.4: Interpreter

2.9 The Loading Pipeline

Before execution, the RVM performs three essential phases: *loading*, *linking*, and *validation*.
Upon invocation, the VM receives a folder path, recursively scans it, and loads all module files Each module is parsed into a corresponding `Module Data` structure.
In the linking phase, the VM allocates and initializes all runtime data buffers, such as constant pools, functions, global variables, and debug information. It also resolves cross-module references by building a `Module Table` for each module.
The final step before execution is *validation*, in which the VM checks that each function

conforms to structural constraints:

- `AWC` is less than or equal to `LWC`.
- The stack pointer remains within $[0, \texttt{SWC}]$.
- All instructions and system calls are valid opcodes.
- Local variable accesses stay within $[0, \texttt{LWC}]$.
- Constant pool accesses do not exceed their bounds.

The only exception is functions that use the `indcall` instruction (see Section 3.11). Since the number of operands pushed or popped is determined dynamically at runtime, these functions cannot be fully validated statically and are assumed valid.
Once loading, linking, and validation complete, the VM looks for the program's entry point, the `Main.main` function. If it is not found, execution is aborted with an error.
Initialization concludes with the creation of a unified `Program Context` structure that holds all runtime metadata:

- Call stack buffer and its upper bound
- Constant word buffer
- Double-Word buffer
- String buffer
- Global variable buffer
- Function buffer
- Module table buffer
- Debug information buffer
- Entry point address (`Main.main`)

These buffers are referenced by the module tables. Grouping runtime data by type improves spatial locality in memory, reducing cache misses and improving execution efficiency.
The interpreter then initializes its internal state using the `Program Context`:

- The Program Counter is set to the entry point's first instruction.
- The Stack Pointer is set to the base of the entry point's operation stack.
- The Frame Pointer is set to the base of the call stack.
- Pool pointers are initialized from the entry point's module table.

After state initialization, the interpreter enters its main execution loop and begins program execution.

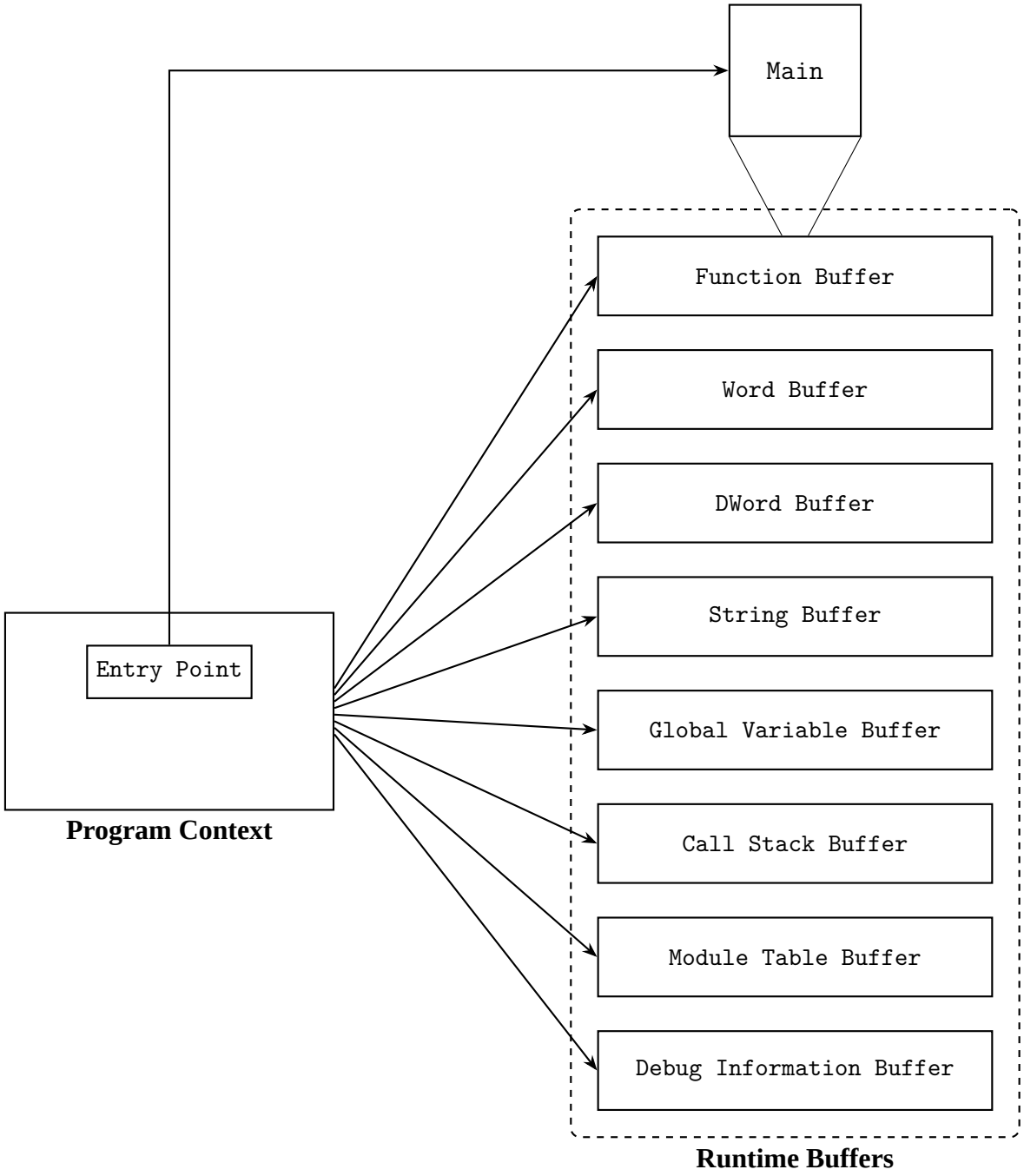

Figure 2.5: Program Context and Program Buffers

Chapter 3

Raiu Instruction Set

In this chapter, I will introduce the Raiu Instruction Set (RIS), the bytecode language that powers the RVM. The chapter will begin by defining the encoding format, byte-sized opcodes with optional static operands, and will explain how dynamic data is passed via the *Operation Stack*. Then it will present the core instruction families; covering data movement, computation, control flow, function calls, and memory operations, each following a uniform naming and parameter convention. By the end of this chapter, the reader will have a clear understanding of the ISA and will therefore be able to design a custom programming language to run on the RVM platform.

3.1 Bytecode

In the RVM, instructions form the fundamental units of execution. Each instruction is identified by an 8-bit unsigned integer opcode, allowing for up to 255 unique operations (opcode 0x0 is considered an invalid opcode). This compact, byte-sized encoding offers simplicity and efficiency, making the instruction format lightweight and easy to decode.

An instruction consists of an opcode and, optionally, a fixed number of static parameters. As a result, instructions may vary in size, unlike traditional machine code, where instruction lengths are often fixed or follow rigid alignment rules.

As discussed previously in Section 2.2.2, the RVM follows a stack-based architecture. Accordingly, dynamic parameters are passed through the Operation Stack. Each instruction consumes its input operands from the stack and pushes its result back onto it.

3.2 Conventions

Before delving into the details of the instruction set, it is essential to establish the conventions used in this document to ensure clarity and consistency in the subsequent presentation of the instructions.
To describe each instruction, I adopted a standardized format, as outlined below.

`family, params; description; (in)` $\rightarrow$ `(out); domains`

Here, the components of this notation are defined as follows:

- `family`: This refers to the instruction family, which groups related instructions together. For instance, an entry such as instr_{v} where $v \in \{0,1\}$ represents the pair of instructions instr_0 and instr_1.
- `params`: This represents the fixed parameters associated with the instruction. These parameters are placed immediately after the instruction in the bytecode.
- `description`: A brief description of the instruction's functionality.
- `in`: This denotes the input parameters that are passed via the Operation Stack to the instruction.
- `out`: These are the output values produced by the instruction, which are subsequently pushed onto the Operation Stack after execution.
- `domains`: This refers to a list of the domains within which the variables involved in the instruction operate.

To illustrate the use of the instruction set in a practical environment, for simplicity, I will provide code examples written in the C programming language. Therefore, a basic understanding of C is recommended to fully understand the use cases presented.

3.2.1 Variables and Their Domains

In the context of the instruction set, variables are values that can assume any value within a specified domain. The domain of a variable dictates a set of values it can take.
The following notations are used to define the properties of variables:

- $t\,X$: This indicates that X is a variable of type t.
- $X \in [a,b]$: This means that X can take any integer value between a and b, inclusive.

- $X \in \{a,b,c\}$: This means that X can assume one of the discrete values a, b, or c.
- some_$\{X\}$: This notation indicates that for every possible value that X can assume within its domain, there exists a corresponding variant denoted by some_X.

These conventions ensure that the structure and behavior of instructions and their associated parameters are well defined and easily understood throughout the document. The use of precise notation allows for an unambiguous representation of the operations and the values they manipulate, which is critical for the subsequent analysis and discussion of the instruction set.

3.3 Push

Every instruction that places a value on top of the Operation Stack is called a push instruction. There are three types of push instructions:

3.3.1 Local Push

Local push instructions differentiate from other push instructions, since they retrieve the value from the local variable array and subsequently push it to the Operation Stack.
In comparison to the JVM, where similar operations are termed ”load instructions”, I have opted to use an alternative nomenclature. This decision stems from the fact that, as will be elaborated in the subsequent sections of this thesis, the RVM also incorporates actual memory load and store operations.
In order to compress data I have also implemented byte and half-word push instructions which are not part of the JVM instruction set; These additions enable the compiler or bytecode programmer to selectively push specific bytes or half-words from a local variable array cell onto the stack.
To show a practical example, let us say we have the following struct.

```
struct MyStruct
{
    i16 x;
    i16 y;
};
```

With the following memory layout.

byte	0	1	2	3
var	x	x	y	y

If we have a stack instance of this struct stored inside local word 0 and we want to access the member variable y we can use the instruction "`push_hword_1, #0`" to place it on the Operation Stack.

- `push_byte_{b}, u8 l`; pushes byte `b` of local word `l`; `()→(word)`; `b` $\in [0,3]$.
- `push_hword_{h}, u8 l`; Pushes hword `h` of local word `l`; `()→(word)`; `h` $\in \{0,1\}$.
- `push_word, u8 l`; Pushes local word `l`; `()→(word)`.
- `push_word_{l}`; Pushes local word `l`; `()→(word)`; `l` $\in [0,3]$.
- `push_dword, u8 l`; Pushes a dword starting from local word `l`; `()→(dword)`.
- `push_dword_{l}`; Pushes a dword starting from local word `l`; `()→(dword)`; `l` $\in [0,3]$.
- `push_words, u8 l, u8 n`; Pushes `n+1` words starting from local word `l`; `()→(word[n+1])`.
- `push_ref, u8 l`; Pushes a reference to local word `l`; `()→(ref)`.

3.3.2 Immediate Push

Immediate push instructions are operations that place a value directly onto the Operation Stack without referencing a constant pool.
In some cases, the value to be pushed is embedded within the instruction itself. For example, there are specific instructions to push the values 0, 1, and 2 for both integer and floating point types.
For larger values, the instruction set includes support for pushing byte-sized integers, which are automatically extended to 32-bit or 64-bit, depending on the context. This provides a compact and efficient way to handle small constants while maintaining flexibility for larger numerical values.

- `push_0_{t}`; Pushes the `t` value 0; `()→(t)`; $t \in \{$`word,dword`$\}$.
- `push_{t}_{v}`; Pushes `v` as a `t` value; `()→(t)`; $v \in \{1,2\}, t \in \{$`i32,i64,f32,f64`$\}$.
- `push{t}, i8 v`; Pushes `v` as a `t` value; `()→(t)`; $t \in \{$`i32, i64`$\}$.

3.3.3 Constant Push

To push immediate values that cannot be represented as small literals, the RVM utilizes constant pools. As described in Chapter 2.3, there are five types of constant pools: word, double word, string and global variables and functions.
Each pool is accessed using two dedicated instructions: one for constants with indices less than 256, and another for constants with indices greater than or equal to 256, up to 2^{16} which is the capacity limit for a constant pool. The latter set of instructions is marked with the ”w” postfix to distinguish them, indicating their ability to handle larger index values.

- `push_const_word, u8 c`; Pushes constant word `c`; `()`→`(word)`.
- `push_const_word_w, u16 c`. Pushes constant word `c`; `()`→`(word)`.
- `push_const_dword, u8 c`; Pushes constant dword `c`; `()`→`(dword)`.
- `push_const_dword_w, u16 c`; Pushes constant dword `c`; `()`→`(dword)`.
- `push_const_str, u8 c`; Pushes a reference to constant string `c`; `()`→`(ref)`.
- `push_const_str_w, u16 c`; Pushes a reference to constant string `c`; `()`→`(ref)`.
- `push_glob_ref, u8 c`; Pushes a reference to global variable `c`; `()`→`(ref)`.
- `push_glob_ref_w, u16 c`; Pushes a reference to global variable `c`; `()`→`(ref)`.
- `push_func, u16 c`; Pushes a reference to a function `c`; `()`→`(ref)`.

3.4 Pop

Pop instructions are the logical opposite of local push instructions, they take a value from the top of the Operation Stack and move it in the Local Scope, those operations are the equivalent of the assign operator on functions local variables.
As discussed earlier in the context of push instructions (see Section 3.3), the JVM refers to a similar family of instructions as ”store instructions”.
In the RVM, the versatility of push operations, which can handle byte and half-word values, is mirrored in the design of pop instructions. Specifically, pop operations can also handle byte-sized and half-word values. In these cases, when a word value is popped from the Operation Stack, it is truncated to fit the target size before being placed in the Local Scope. This automatic truncation ensures compatibility with smaller data types, such as byte and short integers.

This approach contrasts with the JVM, where byte and half-word integers are not directly supported; instead, all integers are treated as 32-bit values. In the JVM, the process of casting smaller integers (like byte and short) must be explicitly invoked before performing the store operation. By contrast, in the virtual machine presented here, such casting is handled automatically during the pop operation, simplifying the programming model, and enhancing the performance of the system.
Let us see an example of a byte truncation:

```
i32 i = 122;
i8 c  = i * 10;
```

In this case, when the operation `i * 10` is performed, the resulting value is `1220`, which in binary is 10010110000_2. Once this value is assigned to `c` using a `pop_byte` instruction, `c` takes the value of the 8 least significant bits, which are 10110000_2, corresponding to the decimal value `-80` in 8-bit signed representation.

- `pop_byte_{b}, u8 l`. Pops to Local Socpe word `t` byte `b`; `(word)`$\rightarrow$`()`; `b` $\in [0, 3]$.
- `pop_hword_{h}, u8 l`; Pops to Local Socpe word `l` hword `h`; `(word)`$\rightarrow$`()`; `h` $\in \{0, 1\}$.
- `pop_word, u8 l`; Pops to Local Scope word `l`; `(word)`$\rightarrow$`()`.
- `pop_word_{l}`; Pops to Local Scope word `l`; `(word)`$\rightarrow$`()`; `l` $\in [0, 3]$.
- `pop_dword, u8 l`; Pops to Local Scope local dword `l`; `(dword)`$\rightarrow$`()`.
- `pop_dword_{l}`; Pops to Local Scope local dword `l`; `(dword)`$\rightarrow$`()`; `l` $\in [0, 3]$.
- `pop_words, u8 l, u8 n`; Pops `n+1` words to Local Scope from local word `l`; `(word[n+1])`$\rightarrow$`()`.

3.5 Arithmetic

All arithmetic instructions, except for negation, require two operands. The RVM does not differentiate between signed and unsigned integers for addition and subtraction; the result is the same in both cases. To maintain consistency, both signed and unsigned addition and subtraction operations are prefixed with ”i” to indicate that they are integer operations.
However, multiplication, division, and reminder operations distinguish between signed and unsigned integers, since the result of those operations may differ depending on the type.

```
i32 z = x * y - 12;
```

```
# x is stored in local word 0
# y is stored in local word 1

push_word_0
push_word_1
mul_i32
push_i32 #12
sub_i32
pop_word_2
```

- `add_{t}`; Adds two `t` values; `(t,t)`→`(t)`; `t` ∈ {`i32,i64,f32,f64`}.
- `inc_{t}, u8 l, u8 v`; Increments by `v` the `t` value from local word `l`; `()`→`()`; `t` ∈ {`i32,i64,f32,f64`}.
- `sub_{t}`; Subtracts two `t` values; `(t,t)`→`(t)`; `t` ∈ `i32,i64,f32,f64`.
- `dec_{t}, u8 l, u8 v`; Decrements by `v` the `t` value from local word `l`; `()`→`()`; `t` ∈ $\{i32, i64, f32, f64\}$.
- `mul_{t}`; Multiply the two `t` values; `(t,t)`→`(t)`; `t` ∈ {`i32,i64,u32,u64,f32,f64`}.
- `div_{t}`; Divide two `t` values; `(t,t)`→`(t)`; `t` ∈ {`i32,i64,u32,u64,f32,f64`}.
- `rem_{t}`; Performs the reminder operation on two `t` values; `(t,t)`→`(t)`; `t` ∈ {`i32,i64,u32,u64`}.
- `neg_{t}`; Negates a t value; `(t)`→`(t)`; `t` ∈ {`i32,i64,f32,f64`}.

3.6 Bitwise

Bitwise operations work directly on individual bits, and in most conventional programming languages, they are typically restricted to integer types. However, I chose to categorize them as ”super type” operations, meaning they can be applied more broadly. The only exception to this

is the right-shift operation, which is exclusive to integer types.
The right-shift operation behaves differently depending on whether the integer is signed or unsigned. Specifically, when performing a right shift on a signed integer, the sign bit is preserved, ensuring that the result maintains the correct sign.

- `and_{t}`; Performs the AND operation on two `t` values; `(t,t)`$\rightarrow$`(t)`; `t` $\in$ {`word,dword`}.
- `or_{t}`; Performs the OR operation on two `t` values; `(t,t)`$\rightarrow$`(t)`; `t` $\in$ {`word,dword`}.
- `xor_{t}`; Performs the exclusive OR operation on two `t` values; `(t,t)`$\rightarrow$`(t)`; `t` $\in$ {`word,dword`}.
- `not_{t}`; Performs the NOT operation on a `t` value; `(t)`$\rightarrow$`(t)`; `t` $\in$ {`word,dword`}.
- `shl_{t}`; Performs the left shift operation a `t` value; `(t,u32)`$\rightarrow$`(t)`; `t` $\in$ {`word,dword`}
- `shr_{t}`; Performs the right shift operation on a `t` value; `(t,u32)`$\rightarrow$`(t)`; `t` $\in \{i32, i64, u32, u64\}$;

3.7 Stack Manipulation

To optimize certain operations, it is essential to have the ability to manipulate stack values using operations like swap and duplicate. In traditional register-based assembly languages, this is not necessary because register values are preserved after an operation is performed. However, the Operation Stack does not retain the operands of an instruction after execution.
Duplicate operations are particularly useful for cases where the same operand is needed multiple times. For example, consider the following operation:

```
i32 z = (x + y) * (x + y);
```

We need the result of the operation `(x + y)` two times, therefore, instead of calculating it twice, it is useful to be able to duplicate it in the Operation Stack.

```
# x is stored in local word 0
# y is stored in local word 1
```

```
push_word_0
push_word_1
add_i32
dup_word
mul_i32
pop_word_2
```

- `dup_{t}`; Duplicates Operation Stack top `t` value; `(t)`→`(t,t)`; `t` ∈ {`word,dword`}
- `dup_{t}_x1`; Duplicates Operation Stack top `t` value and moves the original one word down; `(word,t)`→`(t,word,t)`; `t` ∈ {`word,dword`}.
- `dup_{t}_x2`; Duplicates Operation Stack top `t` value and moves the original two words down; `(dword t)`→`(t,dword,t)`; `t` ∈ {`word,dword`}.
- `swap_{t}`; Swaps Operation Stack top two `t` values; `(t,t)`→`(t,t)`; `t` ∈ {`word,dword`}.

3.8 Cast

Cast instructions do not require static parameters. Instead, they take a value from the Operation Stack and apply a specific type conversion. The valid source-target type pairs supported by the RVM are listed in the following table:

to/from	i8	i16	i32	i64	f32	f64
i32	✓	✓	-	✓	✓	✓
i64	✗	✗	✓	-	✓	✓
f32	✗	✗	✓	✓	-	✓
f64	✗	✗	✓	✓	✓	-

As shown, all 32-bit and 64-bit types support cast operations for every possible pair, excluding, of course, casts between identical types, which are unnecessary.
For extension casts, where the initial type is a byte or half-word integer, only conversions to 32-bit integers are defined. These casts perform sign extension, replicating the sign bit of the smaller type to fill the higher-order bits of the 32-bit result.

- `{t}_to_{i32}`; Converts a `t` value to an i32; `(i32)`→`(t)`; $t \in$ {`i8,i16`}.
- `{t}_to_{u}`; Converts a `t` value to a `u` value; `(t)`→`(u)`; $t,u \in$ {`i32,i64,f32,f64`}, $t \neq u$.

3.9 Comparison

Comparison instructions apply a specified comparison operator by popping the top two elements of a given type from the Operation Stack. The result of the comparison, 1 if the condition is true or 0 if false, is then pushed back onto the Operation Stack as a result word.
The RVM supports the following comparison operations, each identified by a specific suffix:

- `eq`: Equal
- `ne`: Not equal
- `lt`: Less than
- `le`: Less than or equal
- `gt`: Greater than
- `ge`: Greater than or equal

The `eq` and `ne` compare operations do not differentiate between integer and floating point types. In both cases, the comparison checks for exact bitwise equality, meaning all bits must match for the result to be considered true. Because of this, equality comparisons are instead distinguished based on operand size specifically, whether they are word or double-word operations.

- `cmp_{t}_{c}`; Performs the `c` comparison on two `t` values; `(t,t)`$\rightarrow$`(word)`; `t` $\in$ {`word`,`dword`}, `c` $\in$ {`eq`,`ne`}.
- `cmp_{t}_{c}`; Performs the `c` comparison on two `t` values; `(t,t)`$\rightarrow$`(word)`; `t` $\in$ {`i32`,`i64`,`u32`,`u64`,`f32`,`f64`}, `c` $\in$ {`lt`,`le`,`gt`,`ge`}.
- `cmp_not`; Performs the logical NOT operation; `(word)`$\rightarrow$`(word)`.

3.10 Jump

Jump operations can be either conditional or unconditional. Each jump instruction includes a signed half-word integer offset, which indicates the number of bytes to move within the byte-code.
Setting the correct jump offset is critical; an incorrect value may cause the program to jump to unintended locations, leading to unpredictable or erroneous behavior.
Conditional jumps are typically used in combination with comparison instructions, allowing the RVM to implement conditional branching based on runtime values.

```
    if(n < 0)
    {
        // do something
    }
    else
    {
        // do something else
    }
```

```
    # n is stored in local word 0

    # jump if n >= 0
    push_word_0
    push_0_word
    cmp_i32_ge
    jmp_if #ELSE
IF:
    # do something
    jmp #ENDIF
ELSE:
    # do something else
ENDIF:
```

This branching system enables the implementation of common control flow structures, such as loops. Using conditional and unconditional jump instructions in conjunction with comparison operations, it is possible to construct loop types such as for, while, and do-while.
For loops and while loops:

```
    for(i32 i = 0; i < 100; i++)
    {
        // do something
    }
```

```
    i32 i = 0;
    while(t < 100)
    {
         // do something
        i++;
```

```
                                        }
```

```
    push_0_word
    pop_word_0
    jmp #CHECK
BODY:
    # do something
    inc_i32 #0 #1
CHECK:
    push_word_0
    push_i32 #100
    cmp_i32_lt
    jmp_if #BODY
```

Do-while loops are even simpler to implement. Since the condition is evaluated after the loop body, there is no need to jump to the check label before entering the loop body. This ensures that the loop body is executed at least once.

- `jmp, i16 o`; Jumps by `o` bytes; `()`→`()`.
- `jmp_if, i16 o`; Jumps by `o` bytes if Operation Stack top word is not zero; `(word)`→`()`.

3.11 Functions

In the RVM, functions can be invoked in two distinct ways: by calling a user-defined function present in the bytecode, or by invoking a system call provided by the virtual machine itself.
When a user-defined function is called, the number of words specified by the Argument Word Count (see Section 2.4) is popped from the current stack frame's Operation Stack. A new stack frame is then created for the called function, and the argument words are transferred into its Local Scope.
Returning from a function works in the opposite direction. The number of words defined by the RWC are popped from the current Operation Stack and pushed onto the Operation Stack of the caller function. Finally, the current stack frame is discarded, resuming execution in the context of the previous frame.
Here is a simple example of an user-defined function

```
i32 iadd(i32 a, i32 b) { return a + b; }

...

i32 v = iadd(5, 3);
```

```
# name AWC LWC SWC RWC
.fn iadd #2 #2 #2 #1
    push_word_0
    push_word_1
    add_i32
    ret

...

# v is stored in local word 0
push_i32 #5
push_i32 #3
call #iadd
pop_word_0
```

- `call, u16 f`; Calls the function associated to the ID `f`; `(word[AWC])→()`.
- `syscall, u8 f`; Calls the system function associated with the opcode `f`; `(word[params])→()`.
- `indcall`; Calls the function pointed by Operation Stack top reference; `(word[AWC],dword)→()`.
- `ret`; Returns from a function; `(word[RWC])→(word[RWC])`.

3.11.1 System Calls

A system call is a native function of the virtual machine that provides access to special functionalities. These functions are typically either too complex to implement efficiently in bytecode,

such as printing to the console, or require optimized performance, like computing square roots, and are therefore implemented as native C functions.

- `exit`; Exits from the program with a return code; `(word)→()`.
- `print`; Prints a string to standard output; `(ref)→()`.
- `printi`; Prints an `i64` to standard output; `(i64)→()`.
- `printf`; Prints an `f64` to standard output; `(f64)→()`.
- `scan`; Allocates a string containing the next line of the standard input, it's up to the user to free the buffer; `()→(ref)`.
- `scani`; Retrieves the next `i64` value of the standard input; `()→(i64)`.
- `scanf`; Retrieves the next `f64` value of the standard input; `()→(f64)`.
- `memcpy`; Copies a block of memory to another location; `(ref,ref,u32)→()`.
- `memmov`; Moves a block of memory to another location; `(ref,ref,u32)→()`.
- `clock`; Gets the current time in nanoseconds; `()→(u64)`.
- `sqrt32`; Calculates the square root of an `f32` value; `(f32)→(f32)`.
- `sqrt64`; Calculates the square root of an `f64` value; `(f64)→(f64)`.
- `exp32`; Calculates e to the power of an `f32` value; `(f32)→(f32)`.
- `exp64`; Calculates e to the power of an `f64` value; `(f64)→(f64)`.
- `log32`; Calculates the natural logarithm of an `f32` value; `(f32)→(f32)`.
- `log64`; Calculates the natural logarithm of an `f64` value; `(f64)→(f64)`.

3.12 Memory

In contrast to the JVM, memory management in this virtual machine is not automated by a garbage collector; instead, memory management is a responsibility of the programmer. The virtual machine provides two primary functionalities: memory allocation and release, which are handled through specific instructions.
These instructions function similarly to the `malloc()` and `free()` functions in the C programming language, where `malloc()` allocates memory and `free()` releases it.

```
i32 *p = malloc(8 * sizeof(i32));
// do something
free(p);
```

```
# void *p = malloc(8 * sizeof(i32));
push_i32 #32
alloc
pop_dword_0

# do something

# free(p);
push_dword_0
free
```

- `alloc`; Allocates a buffer; `(u32)`→`(ref)`.
- `free`; Frees a block of memory; `(ref)`→`()`.

3.13 Load and Store

Load and store operations are essential for transferring data between memory locations and the Operation Stack. Load instructions retrieve data from memory and push it onto the Operation Stack, while, store instructions perform the reverse operation by taking a value from the stack and storing it at a specified memory location.
To ensure consistency with the design principles, the load and store instructions support operations on byte and half-word data types. This enables the referencing of these smaller data types between stack frames, a capability not available in the JVM.
Load instructions require a reference to the memory location as a dynamic parameter, while store instructions take both a reference and the value to be stored.
There are three distinct types of load and store instructions:

3.13.1 Standard Load and Store

Standard load and store instructions are the simplest type of load and store operation, this kind of instructions treats the parameter pointer as the starting address from which to retrieve or send the block of data.

```
// p is a pointer to an integer
i32 i = *p;

// do something with i

*p = i;
```

```
# p is stored in local words 0 and 1
push_dword_0
load_word
pop_word_2

# do something with local word 2

push_dword_0
push_word_2
store_word
```

- `load_byte_{b}`; Loads byte `b` of the referenced word; `(ref)`$\rightarrow$`(word)`; `b` $\in [0, 3]$;
- `load_hword_{h}`; Loads half word `h` of the referenced word; `(ref)`$\rightarrow$`(word)`; `h` $\in \{0, 1\}$.
- `load_{t}`; Loads a `t` value; `(ref)`$\rightarrow$`(t)`; `t` $\in \{$`word,dword`$\}$.
- `load_words, u8 n`. Loads `n+1` words; `(ref)`$\rightarrow$`(word[n+1])`.
- `store_byte_{b}`; Stores to byte `b` of the word referenced; `(ref,word)`$\rightarrow$`()`; `b` $\in [0, 3]$.
- `store_hword_{h}`; Stores to hword `h` of the word referenced; `(ref,word)`$\rightarrow$`()`; `b` $\in \{0, 1\}$.

- `store_{t}`; Stores a `t` value; `(ref, t)`→`()`; `t` ∈ {`word`,`dword`}.
- `store_words, u8 n`; Stores `n+1` words; `(ref,word[n+1])`→`()`.

3.13.2 Buffer Load and Store

Buffer load and store instructions treat the pointer as a buffer pointer. The maximum allowed size for a buffer element is 256 words (1kB), while the minimum size is one byte.
These instructions require an additional parameter, which is passed via the Operation Stack: an u32 value representing the index of the element to be accessed.
When working with buffers, we usually want to work with references to the buffer elements. To simplify these operations, I also implemented instructions that load references to a buffer element. To get the address of an element, the formula for calculating it is as follows:

```
T *e = buffer + index * sizeof(T);
```

Performing this operation without a specialized instruction results in the following code block:

```
# buffer is stored at local words 0 and 1
# index is stored at local word 2
# sizeof(T) = 4

push_dword_0
push_word_2
i32_to_i64
push_i64 #4 # sizeof(T)
mul_i64
add_i64
pop_dword_3
```

It becomes evident that this frequently executed operation incurs a high instruction overhead; however, introducing a reference load instruction significantly reduces the overall instruction count.

```
push_dword_0
push_word_2
load_buff_word_ref
pop_dword_3
```

Every time a buffer is accessed the validity of the operation is checked and if the memory has been accessed outside of its bounds the program terminates with a ”BufferOverflowError” message.

- `load_buff_{t}_val`; Loads the value of an element of a `t` buffer; `(ref, u32)`$\rightarrow$`(t)`; `t` $\in$ {`byte,hword,word,dword`}.
- `load_buff_words_val, u8 n`; Loads an element of a buffer with elements of `n+1` words; `(ref,u32)`$\rightarrow$`(word[n+1])`.
- `load_buff_{t}_ref`; Loads a reference to an element of a `t` buffer; `(ref,u32)`$\rightarrow$`(ref)`; `t` $\in$ {`byte,hword,word,dword`}.
- `load_buff_words_ref, u8 n`; Loads a reference to a buffer with elements of `n+1` words; `(ref,u32)`$\rightarrow$`(ref)`.
- `store_buff_{t}`; Stores an element of a `t` buffer; `(ref,u32,t)`$\rightarrow$`()`; `t` $\in$ {`byte,hword,word,dword`}.
- `store_buff_words, u8 n`; Stores an element of a buffer with elements of `n+1` words; `(ref,u32,word[n+1])`$\rightarrow$`()`.

3.13.3 Offsetted Load and Store

In order to facilitate operations on data structures, support for offset-based load and store instructions is essential. Consistent with the overall design of the RVM, I chose to implement fixed 8-bit offsets ranging from 0 to 255, embedded directly as parameters within the bytecode. The primary use case for these instructions lies in type grouping. As outlined in the preceding sections, the RVM intentionally omits the use of type metatables, a mechanism commonly adopted by the JVM and the CLR.
While this design decision imposes a limitation on data structure size, restricting them to a maximum of 1kB (256 words), and results in increased bytecode footprint, it is motivated by the potential for improved runtime performance and reduced system complexity.
Let us look at a practical example on how these instructions may be used:

```
struct vec2
{
    float x;
    float y;
    float z;
};
```

This is the memory layout of the struct.

byte	0	1	2	3	4	5	6	7	8	9	10	11
var	x	x	x	x	y	y	y	y	z	z	z	z

```
// data is a pointer to a Data struct
data->z = data->x + data->y;
```

```
# data is stored in local words 0 and 1
push_dword_0
push_dword_0
load_word     # vec2::x
push_dword_0
load_ofst_word #1  # vec2::y
add_f32
store_ofst_word #2 # vec2::z
```

- `load_ofst_byte_{b}, u8 o`; Loads byte `b` of the word referenced offsetted by `o` words; `(ref)`→`(word)`; `b` $\in [0, 3]$.
- `load_ofst_hword_{h}, u8 o`; Loads hword `h` of the word referenced offsetted by `o` words; `(ref)`→`(word)`; `h` $\in \{0, 1\}$.
- `load_ofst_{t}, u8 o`; Loads a `t` value offsetted by `o` words; `(ref)`→`t`; `t` $\in$ {`word, dword`}.
- `load_ofst_words, u8 o, u8 n`; Loads `n+1` words from offsetted reference; `(ref)`→`(word[n+1])`.

- `store_ofst_byte_{b}, u8 o`; Stores to byte `b` of the referenced word offsetted by `o` words; `(ref,word)`$\rightarrow$`()`; `b` $\in [0, 3]$.
- `store_ofst_hword_{h}, u8 o`; Stores to hword `h` of the referenced word offsetted by `o` words; `(ref,word)`$\rightarrow$`()`; `h` $\in 0, 1$.
- `store_ofst_{t}, u8 o`; Stores a `t` value offsetted by `o` words; `(ref,t)`$\rightarrow$`()`; `t` $\in$ {`word, dword`}.
- `store_ofst_words, u8 o, u8 n`; Stores `n+1` words to offsetted reference; `(ref,word[n+1])`$\rightarrow$`()`.

3.14 Unsafe Behavior

Before concluding, it is important to highlight a fundamental design decision of the RVM: unlike established virtual machines such as the JVM or the CLR, the RVM does not prioritize strong runtime safety guarantees. This choice was made deliberately to avoid the performance overhead associated with constant validity checks, particularly in memory management.
The RVM provides only a minimal level of protection against unsafe or malformed bytecode. As such, it is crucial to define what constitutes *defined or undefined unsafe operations* within this execution model.

3.14.1 Checked Unsafe Operations

These operations are detected and prevented either at load time, by the code validator, or during execution, by the interpreter:

- **Static validation failures (load time):**
  - Pushing to or popping from a local variable index out of scope.
  - Accessing a constant pool with an invalid index.
  - Underflowing or overflowing the Operation Stack.
  - Invoking a invalid function or system call.
  - Failing to return from a function.
  - Jumping with an invalid offset.

- **Dynamic checks (runtime):**
  - Call Stack overflow.
  - Buffer bounds violations (any access outside the declared buffer length).

When a checked unsafe operation is encountered, the VM halts execution and reports an error, preventing further corruption.

3.14.2 Unchecked Unsafe Operations

The following operations are not automatically validated and therefore may cause unpredictable behavior, memory corruption, or immediate crashes:

- **Accessing invalid pointers:** Accessing invalid addresses can result in memory corruption or crashes, depending on the memory region affected.
- **Using incorrect offsets on valid pointers:** Accessing memory using a valid base address but an incorrect offset can also result in corruption or crashes.
- **Freeing invalid pointers:** Attempting to deallocate memory using a pointer that was not previously allocated (or was already freed) may corrupt the internal allocator, potentially leading to a crash on subsequent allocations or deallocations.
- **Calling invalid function pointers:** If the VM attempts to execute code from an address that does not point to a valid function, it may interpret arbitrary memory as instructions. This can result in erratic behavior or immediate failure.

Most of these issues can be prevented by well-designed compilers or languages via type safety and strict pointer discipline. Null pointer dereferences remain a challenge but can be mitigated at compile time with null safety features.
Finally, in the current implementation a memory access violation causes the underlying operating system to terminate the RVM process. Future versions will intercept such faults, raise structured errors, and display diagnostic information, including the offending instruction and relevant VM state, to improve debuggability and resilience.

Chapter 4

Conclusions

4.1 Challenges

The development of this project involved several significant challenges inherent to virtual machine design. What began as a hobby project aimed at testing and improving programming skills quickly revealed the complexity involved in creating an efficient virtual machine, together with a coherent and extensible instruction set. Designing a VM requires careful consideration of many interdependent components, including memory management, instruction set architecture, execution flow, and resource handling. Balancing performance with simplicity and extensibility proved to be particularly difficult. One key insight from this process is the critical importance of thorough planning and architectural design before implementation, this stems from the fact that rushed development usually results in costly redesigns. The current structure of the RVM reflects several months of iterative development, during which multiple design approaches were explored, evaluated, and refined.

4.2 Improvements

In its current state, the RVM is not yet capable of competing with mature virtual machines. However, with targeted enhancements, which i will explore in this section, the RVM has the potential to evolve into a viable and competitive platform for running programs.

4.2.1 Wide Instructions

One limitation of the current RIS design arises from its 8-bit opcode encoding, which restricts the total number of possible opcodes to 256. As a result, it is not feasible to define a dedicated wide variant for every instruction that may require larger operands.

To address this, the RIS can incorporate a special meta-instruction called `wide`. When encountered, this instruction modifies the behavior of the following instruction by expanding the size of its static operands. Specifically, the operand widths are doubled, allowing access to extended ranges and indices beyond the default limits.
For example, to push a local word at index 1000 onto the stack using a wide instruction, the bytecode sequence would be:

```
(u8)wide, (u8)push_word, (u16)1000
```

This mechanism allows the RVM to lift several architectural limitations:

- Support for stack frames with local variable arrays of up to 256kB in size ($2^{16} \times 4$ bytes).
- Expansion of the addressable memory range for objects up to 256kB.
- Extension of branch instructions to cover jump offsets in the full 32-bit signed range ($[-2^{31}, 2^{31} - 1]$), accommodating arbitrarily large function bodies or code blocks.

This design is inspired by similar mechanisms in mature virtual machines such as the JVM, where a single wide prefix serves to generalize operand size extension without requiring separate opcodes for each wide variant. This keeps the instruction set compact while allowing for future scalability and expressiveness.

4.2.2 Native Method Support

A common technique to boost performance in interpreted languages is to offload certain tasks to low-level compiled code, written in languages such as C, C++, or Rust. To facilitate this, many high-level languages offer a mechanism for calling native functions through a dedicated native interface.
One challenge in supporting native calls is the variability in *Application Binary Interfaces* (ABIs) across platforms and languages. An ABI defines how programs and libraries interact at the binary level and ensures compatibility between independently compiled components.
A typical ABI specifies:

- Calling convention: How function parameters are passed (e.g., via registers or stack).
- Register usage: Which registers must be preserved by the caller or callee.

- Data layout: How data structures are aligned and represented in memory.
- Stack alignment: The required alignment of the stack before a function call.

To support native methods in the Raiu Virtual Machine, I plan to implement a native interface similar to the *Java Native Interface* (JNI) [2]. Initially, the standard ABI chosen for native interoperability is the *C ABI*, as it is widely supported by many systems and languages, including C++, Rust, and others.
To bind a native function, a naming convention is adopted that encodes the package, file, and function name into a C-compatible symbol. Each symbol begins with the prefix `"Raiu"` and then follow the path from the package to the function name, with components separated by underscores.
For instance, a Raiu function with the signature `Package/Folder/File.Foo(i32,i32)(i32)` becomes the C symbol:

```
i32 Raiu_Package_Folder_File_Foo(i32 a, i32 b) {
    // implementation
}
```

Must be noticed how the function signature standard must be modified adding the input and output parameters into the function name, this way the RVM will be able to understand how to pass arguments to the native function.
For example let us take the C ABI for x86-64 processors on linux systems [5], using the previous function the VM must first extract two words from the Operation Stack, and then move the first word into the register `rdi` and the second into register `rsi`.
This approach ensures that the virtual machine can locate and invoke native implementations consistently across different platforms and compilers, while adhering to a portable and stable ABI.

4.2.3 JIT and AOT Compiler

Throughout this thesis, I have frequently referenced the JIT compiler, which is an essential component for most modern interpreted languages. Some Python interpreters, for example, also implement JIT compilation. Another important component in the compilation process is the

Ahead-Of-Time (AOT) compiler.
The distinction between JIT and AOT compilers lies in their respective compilation timings. An AOT compiler performs compilation before code execution, generating machine code before the program starts running. In contrast, a JIT compiler analyzes the code at runtime and compiles the ”hottest” or most frequently executed code paths dynamically, optimizing them for performance as the program runs.
Implementing such a system is a highly complex task for several reasons.

- Platform and OS Dependencies: Each supported platform and operating system requires its implementation of a JIT and AOT compiler.
- Register-based Assembly Languages: Assembly languages are register-based, meaning that the arguments passed to functions are not consumed immediately during instruction execution but may be used for future operations. This introduces additional considerations for the compiler’s behavior.
- Code Optimization: The compiler must be capable of optimizing the compiled code to enhance performance.

Due to the enormous world of code optimization strategies, I will not dive into details here. However, it is worth noting that the JVM runtime employs a two-level compilation strategy. The first level performs a simple and fast translation of bytecode into machine code. The second level involves more aggressive optimizations, but with the downside of slower compilation time. As a result, only the ”hot” code paths, the most frequently executed parts of the program, are subjected to this higher level of optimization.
To illustrate this concept, consider the following code block:

```
i32 counter = 0;
for(i32 i = 0; i < 1000000000; i++)
    counter++;
```

It is evident that this loop, iterating one billion times, is wasteful. A JIT compiler would likely optimize this code by replacing the loop with a direct assignment to the variable counter, as shown below:

```
i32 counter = 1000000000;
```

This transformation reduces both runtime overhead and the amount of unnecessary code execution.

4.3 The Final Result

This thesis has delivered a fully functional Raiu Virtual Machine together with the Raiu Instruction Set. The RVM executes terminal-based programs using compact bytecode instructions and demonstrates core VM features, function invocation, manual memory management, constant pools, and basic I/O, aligned to hardware architectures.
Despite deliberately omitting complex runtime features such as garbage collection and JIT compilation, the implementation supports applications of moderate complexity while maintaining a low overhead. Key achievements include:

- A minimal yet expressive type and memory model, encompassing primitive data types, composite types and buffers.
- An efficient Call Stack and Operation Stack mechanism that underlies function calls, argument passing, and temporary value management.
- A direct threaded interpreter engine with a 8-bit opcode encoding enabling fast decode and dispatch.
- A robust constant pool system for literals, globals, and function references that scales to thousands of entries.
- Basic safety checks, static validation, stack bounds enforcement, and buffer bounds checking, guaranteeing a certain level of runtime safety.

Taken together, these components form a cohesive platform that validates the feasibility of a lightweight, and performant VM. The RVM serves not only as a proof of concept but also as a foundation for future extensions, such as wider instruction operands, native method support, JIT/AOT compilation, and enhanced exception handling, enabling the RVM to compete with established and mature virtual machines.
In conclusion, this thesis has explored a wide range of technical aspects such as memory management, bytecode structure, and function handling, providing valuable information on low-level system design.
The complete source code for this project are available in the Raiu GitHub repository, providing

a basis for continued development and open source collaboration.

Bibliography

[1] William Kahan. “IEEE standard 754 for binary floating-point arithmetic”. In: *Lecture Notes on the Status of IEEE* 754.94720-1776 (1996), p. 11.

[2] Sheng Liang. *The Java native interface: programmer’s guide and specification*. Addison-Wesley Professional, 1999.

[3] Serge Lidin. *.NET IL Assembler*. Apress, 2014.

[4] Tim Lindholm et al. *The Java virtual machine specification*. Addison-wesley, 2013.

[5] HJ Lu et al. “System V application binary interface”. In: *AMD64 architecture processor supplement* (2018), pp. 588–601.

[6] AP Thijssen and HA Vink. “Two’s complement arithmetic”. In: *The seventh international conference Electronics*. Vol. 98. 1998, pp. 150–156.

Acknowledgement

I would like to thank my supervisor, Professor Francesco Silvestri for his guidance throughout the writing of this thesis.
A special thanks goes to my family, who have always supported me, and to my friends who have filled these years with happy and memorable moments.
I think this journey has significantly expanded my technical skill set, and I will look back on it as a truly meaningful and unforgettable experience.